The Virtual Rendering System - a Toolkit for Object-Oriented 3D Graphics

Juergen Doellner Klaus Hinrichs

FB 15, Informatik, Westfälische Wilhelms-Universität
Einsteinstr. 62, D - 48149 Münster, Germany
Phone & FAX: (++49) 251 / 83 - 3755; email: {dollner, khh}@math.uni-muenster.de

Abstract

3D applications are built on top of procedural low-level graphics packages which are difficult to learn and to use because of their inherent complexity and their renderer oriented design. We present a fine-grained object oriented model which views 3D graphics from the developer's perspective. Our approach is based on a logical decomposition of the elements of 3D graphics into three major classes: *Geometric primitives* define shapes and their geometry. *Rendering attributes* specify quality and appearance of primitives and of the rendering process. *Virtual rendering devices* process attributes and primitives through a set of generic rendering commands for different types of rendering techniques and packages. Virtual rendering devices encapsulate the functionality of most of today's graphics packages making them exchangeable even at runtime without the need to recode the application. We have implemented our concepts in VRS, the *Virtual Rendering System*, as a portable C++ toolkit. Currently we have integrated the standard graphics packages OpenGL, PEX, XGL, and Radiance.

Keywords: object-oriented 3D graphics, graphics software architecture, rendering techniques.

1 Introduction

3D graphics packages are increasingly important for the development of multimedia and virtual reality applications. On the one hand new rendering techniques allow to create photorealistic scenes, and fast high-performance graphics hardware enables us to render even complex scenes in real time. On the other hand developers of such applications are confronted with low-level graphics systems which still rely on the procedural programming paradigm. This leads to the following problems:

- Graphics packages are difficult to learn and hard to use because of the multitude of data structures and subroutines the application developer has to understand before he can use these packages. In particular, interfaces of graphics packages are oriented towards the rendering pipeline instead towards the developer's perspective, and they provide only a low level of abstraction. Even for common graphics applications it requires a lot of experience to work efficiently with these libraries.
- Graphics packages are not fine-grained object-oriented, most packages are not object-oriented at all. Object-oriented applications take full advantage of this programming paradigm only if its components are object-oriented, too.
- Graphics applications are not portable across different graphics platforms. To port applications to new graphics platforms, they must be redesigned completely. For instance, applications based on an immediate-mode library cannot be ported easily to radiosity-based packages.

In general, we cannot expect to write a new object-oriented graphics system since standard libraries contain a lot of well designed algorithms and access graphics hardware which is generally hard to program. Therefore, we have to seek for an object-oriented framework which embeds these standard libraries in such a way that we can take advantage of key object-oriented techniques such as classes, encapsulation, polymorphism, and inheritance.

We present an object-oriented framework which generalizes the concepts of most of today's graphics packages. The VRS, the *Virtual Rendering System*, provides an extensible class hierarchy which defines geometric primitives, geometric transformations, rendering attributes, and virtual rendering devices. Primitives, attributes, and transformations are processed by virtual rendering devices which define a homogenous interface for rendering commands. VRS offers easy access to complex capabilities of non-object-oriented graphics systems, and adds extensibility to these packages. We have implemented VRS as a C++ toolkit which embeds the standard low-level graphics systems OpenGL [13], PEX [9], XGL [10], and the physically-based lighting and simulation system Radiance [12]. Other packages are currently integrated.

2 Layers of Graphical Applications

Graphics applications can be divided logically into three layers: rendering, modeling, and the application layers (Fig. 1). Application designers normally interact with the modeling layer, whereas developers of new rendering features interact with the rendering layer.

Application Layer

The application layer handles application objects which encapsulate all the relevant information of the application, e.g. geographical 3D applications maintain landscape objects (Fig. 1). Information stored in application objects is used by the modeling layer for visualization.

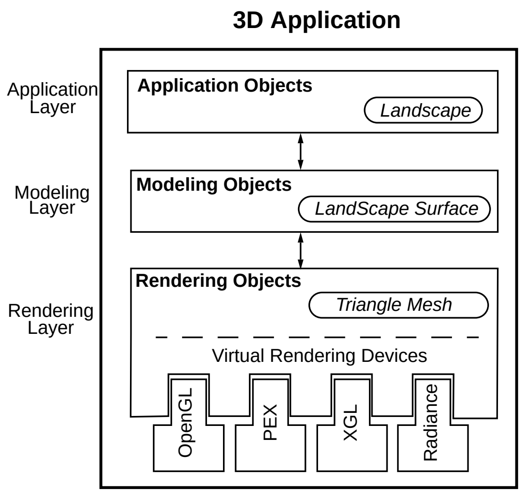


Figure 1: 3D Application Layers

Modeling Layer

The modeling layer mediates between the application layer and the rendering layer. It extracts and interprets information provided by the application layer and transforms the information into objects suitable for the rendering layer. For visualization and animation purposes the application has to assign each application object modeling objects which may be composed of simpler modeling objects in a hierarchical manner. However, the application does not have to care about rendering. The application instructs the modeling objects to render themselves. In contrast to application objects, modeling objects only contain information needed for the visualization, and depend on application objects. The modeling and the application layer may be tightly coupled. In our example, landscape surface descriptions are extracted from landscape objects. These surface descriptions contain geometric coordinates and surface properties.

Rendering Layer

The rendering layer provides the interface to an underlying rendering package. Modeling objects use rendering objects as “assembler language” for 3D graphics. In traditional applications, data structures and subroutines of rendering packages are used in this layer. The rendering layer is critical because it strongly depends on the used package, and it is difficult to program even for experienced programmers. If applications use two or more different rendering packages, the implementation gets more complicated because detailed knowledge about the individual rendering packages must be integrated, and code can rarely be shared. In the example, landscape surfaces are implemented with triangle mesh objects of the rendering layer. Virtual rendering devices render triangle meshes; however, the modeling layer does not need to know how this rendering is performed.

Why separating the application in different layers?

Dividing the system into three major layers offers several advantages: Different renderers can be used by the application to meet different needs. For example, fast low-quality renderers may be used to set up scenes interactively, but the final image generation can be done by high-quality renderers.

Exchangeability of renderers guarantees better portability of applications between different hardware and software platforms.

Application designers benefit from the separation because they are not involved in the complex handling of low-level graphics packages. They can concentrate on building modeling and application objects. Modeling objects delegate most of the graphical operations to well-defined abstract graphical data types, the rendering objects, which are provided by the rendering layer. Furthermore, the rendering layer can be extended by integrating other rendering packages, new rendering primitives, or new rendering attributes. Existing rendering classes can therefore be subclassed.

Traditional graphics systems such as PHIGS [6] or HOOPS [15] do not separate modeling and rendering. They provide a high-level application programming interface which, in general, is neither efficient nor suitable for all kinds of graphical applications. For example, HOOPS cannot build new types of primitives or attributes because it does not provide a class interface. PHIGS cannot be used efficiently for interactive or time-critical applications due to the overhead of maintaining an internal database. In contrast to PHIGS, OpenGL provides 3D hardware and rendering pipeline oriented low-level commands which are generally to elementary to be used by application designers. The VRS encapsulation for OpenGL, however, raises the level of abstraction and maintains full performance.

Only an object oriented and extensible rendering system can serve as base for different modeling layers. To achieve this, we define abstract graphical data types which are processed through a simple pipeline model in virtual rendering devices.

3 Virtual Rendering System

VRS defines three object categories: geometric primitives, rendering attributes, and virtual rendering devices. *Geometric primitives* store geometric shape information. *Rendering attributes* specify the characteristics of geometric primitives and scenes, and specify coordinate systems through geometric transformations. *Virtual rendering devices* draw three-dimensional compositions of primitives called *scenes*. The core classes of VRS are shown in Fig. 2 (given in OMT notation [16]).

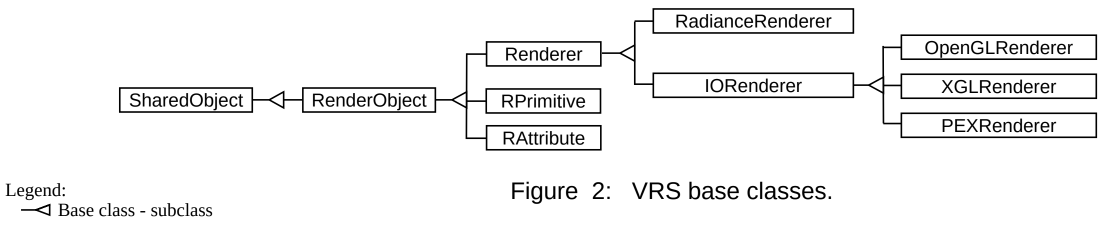


Figure 2: VRS base classes.

3.1 Virtual Rendering Devices

Virtual rendering devices generate scenes by processing primitive and attribute objects based on only three operations: push and pop attributes, and render primitives. The *rendering protocol* defines this set of generic operations which are understood by all virtual rendering devices understand (Fig. 3). They can be grouped in scene management, attribute management, rendering of primitives, and capability information.

```
class Renderer : public RenderObject {
public:
virtual void begin_rendering();
virtual void end_rendering();

virtual void push(RAttribute*, float priority=0.0);
virtual void pop(RAttribute*);

virtual void render(RPrimitive*);

virtual Table<RCapability> supported_primitives();
virtual Table<RCapability> supported_attributes();
 ...
};
```

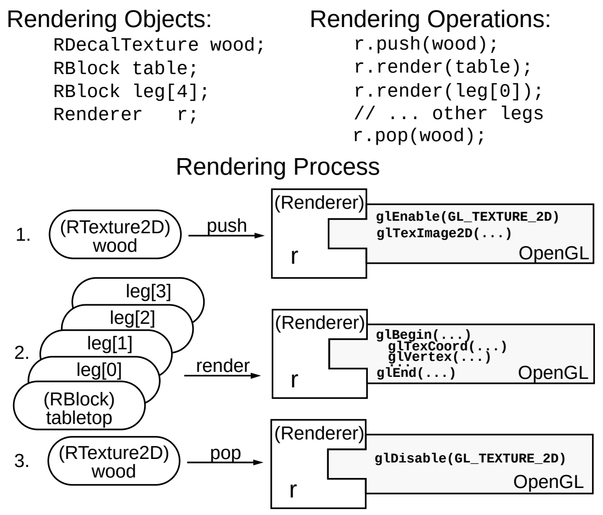
Figure 3: Rendering Protocol.

Rendering devices manage attributes with priority stacks (one for each attribute class). *Priority stacks* provide access to the top element and to the element with the highest priority currently in the stack. The priority is specified when the attribute is pushed. Pushing an attribute changes the current value of that attribute type only if there is no attribute of the same type with higher priority. If primitives are drawn, rendering devices apply to them all relevant attribute values having highest priority.

Rendering primitives are drawn by the render operation. In contrast to [4], we do not include transformation matrices in primitives; transformations are modeled by the 'current modeling and transformation matrix' attribute. Due to that we can share primitives instead of generating multiple instances.

3D applications typically maintain and manipulate a set of rendering objects. Virtual rendering devices map rendering objects to appropriate data structures and commands of the underlying graphics package. This indirection hides all rendering technique specific tasks and leads to a uniform rendering language.

Fig. 4 shows how to represent and how to render a wood-texture table. *RDecalTexture* objects are texture attributes, *RBlocks* are 3D parallelepipeds. Virtual rendering devices are objects of type *Renderer*. In the example, the rendering objects are sent to an OpenGL rendering device.


Rendering Objects:
```
RDecalTexture wood;
RBlock table;
RBlock leg[4];
Renderer   r;
```
Rendering Operations:
```
r.push(wood);
r.render(table);
r.render(leg[0]);
// ... other legs
r.pop(wood);
```
Rendering Process

1.
```
glEnable(GL_TEXTURE_2D)
glTexImage2D(...)
```
2.
```
glBegin(...)
  glTexCoord(...)
  glVertex(...)
  ...
glEnd(...)
```
3.
```
glDisable(GL_TEXTURE_2D)
```

Figure 4: Processing of Rendering Objects through Virtual Rendering Devices

Capability Information

The application can base its decision how to render modeling objects upon the capabilities of renderers given by the *primitive capability table* and the *attribute capability table*. Table entries contain the class name and the rendering efficiency of primitives and attributes. They inform which primitives and attributes are supported, and which of them are accelerated. Renderer specific features, e.g. other primitives or additional attributes, can be inquired of the capability table.

As in [2] a modeling object can provide an object conversion table. Its entries describe the primitives to which the modeling object can be converted. The optimal rendering primitive to be used is the one with the highest value obtained when multiplying the conversion priority with the rendering efficiency.

Rendering devices guarantee that all VRS rendering primitives can be drawn. Those primitives which cannot be mapped directly to primitives of the underlying rendering package are converted into lower-level primitives. Attributes, however, are evaluated only to the degree possible for a specific rendering device. Nevertheless, all attributes can be passed to all rendering devices.

Interaction

Input/output rendering devices support selection tests for primitives. To inquire which primitive objects are inside a view plane area (e.g. an area around the cursor position) a hit area object is sent to the rendering device. All primitives intersecting the hit area are stored in a pick path object. A pick path contains pick information for all primitives which intersect the hit area. The pick information consists of references to these primitive objects. Additionally, the pick information contains a list of all pick name attribute objects which have been pushed when the hit occurred.

3.2 Rendering Primitives

The class hierarchy (Fig. 5) of VRS is based on a logical decomposition of graphical 3D primitives. It distinguishes between elementary primitives (e. g. points, lines), vertex-based primitives (e. g. triangles, quadrilaterals), analytic primitives (e. g. spheres, cones), and surfaces (e. g. NURBS).

In object-oriented graphics systems the class hierarchy for primitives is crucial since it determines both efficiency and reusability of the system. Class hierarchies often suffer from the following disadvantages:

- Classes are inherited to share code although the derived classes are not true subtypes of their parent classes. If the commonality in the internal representation is used as discriminator for inheritance (as in [2] and [4]), we get semantic inconsistencies which must be resolved by blocking operations of parent classes. For example, GRAMS derives boxes from parallelepipeds, and cubes from boxes, but operations of parent classes contradict operations of derived classes, e.g. operations setting width, height, and length derived from parallelepipeds cannot be applied to cubes.
- Primitives assume how rendering devices represent them. For example, GROOP [4] derives cones from triangle meshes. Cones *may* be represented by triangle meshes but some rendering devices can represent them directly. This leads to redundancy in the object representation and affects efficiency. If objects assume how they are represented, and if they maintain this representation, they are not lightweight enough to be used as abstract graphical data types in the whole application, particularly if they are used in large numbers. Additionally, high quality rendering packages even accept analytic primitives in their parametric form.
- The appearance of primitives is defined by a fixed set of attributes stored as object data. However, objects in general share most attributes. So, storing attributes as separated shared objects dramatically reduces the space requirements of complex scenes. Additionally, application specific attribute types can be applied to primitives.

The VRS primitive class hierarchy focuses on a redundancy free and logically decomposed representation of commonly available 3D primitives. We do not include higher-level graphics objects in these primitive core classes. For example, particle systems may be implemented by point sets; in the VRS concept, however, particle systems are considered as modeling objects which aggregate point sets as object data.

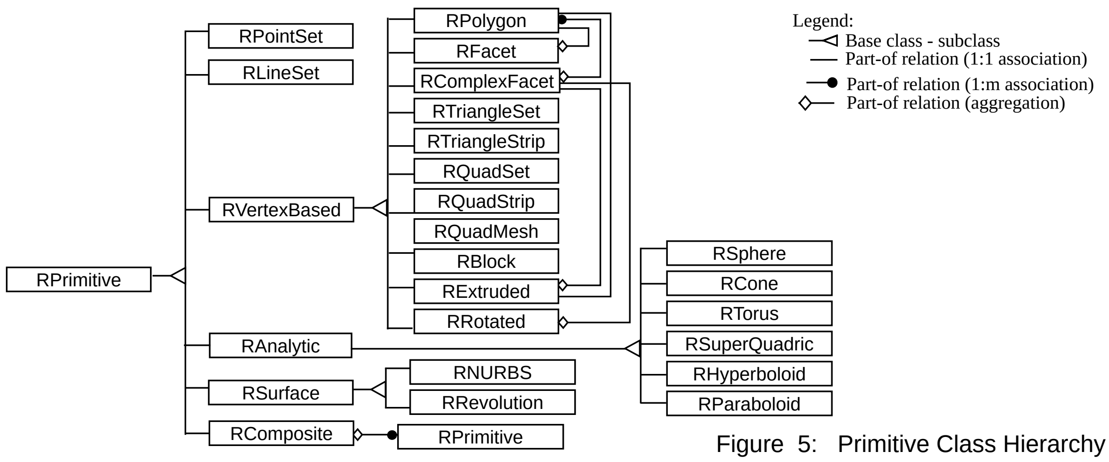


Figure 5: Primitive Class Hierarchy

Vertex-Based Primitives

A vertex-based primitive is characterized by its vertices and the data associated with each vertex, e. g. vertex normals, vertex colors, vertex texture coordinates. Polygons describe a sequence of connected line segments given by a list of vertices. Facets are defined by its planar boundary polygon. Complex facets consist of a set of boundary loops. The interior of a complex facet is determined by the odd-winding rule. Facets and complex facets aggregate polygon objects. Complex facets can be extruded by sweeping the facet along a polygonal path. Rotational vertex-based primitives are created by rotating a polygonal contour. The tessellation of these primitives is determined by the tessellation attribute.

Polygons are reused to a high degree which simplifies both the usage as well as the implementation of vertex-based primitives. For example, extruded (resp. lofted) objects are built by defining a close polygon for the contour and a polyline for the sweep path. To extrude a given facet, we build an extruded primitive passing the boundary polygon of the facet together with a sweep polyline (Fig. 6). Analogously, polygonal boundaries can be swept rotationally.

```
float coordinates[] = { ... };

RPolygon contour(10, coordinates);

RFacet planar_letter(contour);

RExtruded lofted_letter(
  planer_letter, Vector(0,0,-0.2)
);

RRotation rotated_letter(
  planar_letter, 0, 45
);
```

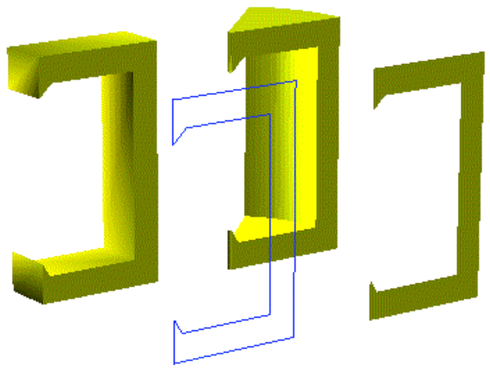


Figure 6: Sweeping Polygonial Contours.

Other vertex primitives are triangle sets (sets of independent triangles), triangle strips (bands or fans of adjacent triangles), quadrilateral sets (sets of independent quadrilaterals), quadrilateral strips and quadrilateral meshes (sets of adjacent quadrilaterals). We distinguish these specialized facet sets to reduce storage overhead and to take advantage of hardware acceleration which is typically provided for triangle and quadrilateral based primitives.

Analytic Primitives

Analytic primitives include spheres, tori, cones, superquadrics, hyperboloids, and paraboloids. They are defined by their analytic parameters and make no assumption about how rendering devices visualize them. For instance, OpenGL and PEX generate triangular meshes for spheres. XGL uses triangular meshes too, but can improve the lighting calculation. Other renderers, e.g. Radiance, accept the analytic description.

3.3 Rendering Attributes

Attributes modify the appearance and geometry of rendering primitives. Attributes encompass surface characteristics (e. g. reflection coefficients, light emission, and textures), representation quality (e.g. tessellation and shading), styles for edges and facets, and attributes applied to the whole scene, e.g. fog and ambient light. VRS designs attributes as separate objects which are processed by rendering devices like primitives.

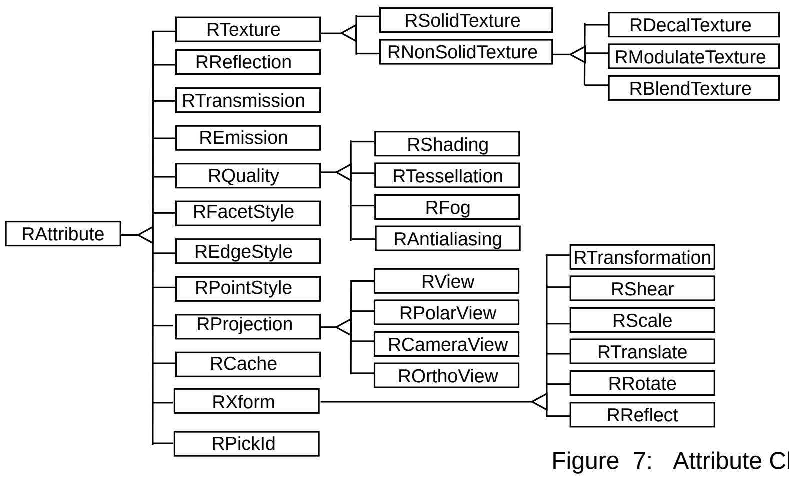


Figure 7: Attribute Class Hierarchy.

We opt for this concept

- to provide a hierarchy of standardized attributes for different rendering packages and techniques,
- to decouple attribute and primitive management, and
- to provide base classes for new attributes or rendering device specific attributes.

The attribute hierarchy (Fig. 7) raises the level of abstraction and provides a standardized access to commonly available attributes. For example, solid texture (i.e. solid colors) attributes apply to points as well as to lines, facets, or analytic primitives. PEX, however, distinguishes between edge color, line color and "interior" color (for facets). Rendering devices evaluate attributes in a context-sensitive mode, i.e. depending on the type of the primitive they decide how to map an attribute to the underlying low-level graphics package.

Attributes can be created and manipulated independently from rendering primitives. Also, attributes can be shared among a group of primitives. Concentrating the attribute management in attribute classes raises the level of abstraction because logically-dependent attribute values are represented together. For example, edge style attributes group together the edge width, the edge visibility flag, and the edge color, whereas in PEX all these values represent individual attributes. Object-oriented encapsulation simplifies the usage of attributes.

The attribute class hierarchy serves as base for new attribute classes. Due to that, VRS allows to access rendering package specific features through the homogenous rendering protocol. Rendering devices determine through runtime type information to which class attribute objects belong, and decide how to evaluate these attribute objects. Particularly, high-performance or high-quality rendering devices provide their own attribute classes. No static attribute hierarchy could cover all possible rendering features. For example, XGL additionally defines for edges antialiasing, join styles and cap styles. The XGL edge style class is derived from the core edge style class. Only XGL rendering devices interpret this attribute class; all other rendering devices treat XGL edge style objects as core edge style objects.

Attribute Priorities

Rendering devices decide which of the available attribute types they apply to rendering primitives. Attributes which cannot be evaluated by the rendering device are ignored. Attribute objects are assigned priorities. For each evaluated attribute type, rendering devices assign that attribute object with the highest priority. Priorities for attribute objects offer additional control over their global influence. Particularly, if we want to overwrite attribute objects defined in subparts of a rendering command sequence, we can conceal these attribute objects through higher priority. For example, to get a wire-framed scene, a drawing style attribute object with infinite priority is pushed at the beginning of the rendering command sequence.

Light Emission Attribute

Lightsources are modeled by primitives emitting light. The light emission depends on the primitive's shape. Emission attributes correspond to physical concepts. From the developer's perspective, it is more natural to think of physical lightsources as real objects instead of abstract lightsources without geometry. In general, emission attribute objects are bound to rendering primitives. The primitive's position determines the position of the lightsource and its emission form. Only ambient light emissions are not bound to rendering primitives.

Rendering devices decide how to map light emissions to primitives. Immediate-mode libraries can map emission attribute objects based on the shape to which the emission is applied. Cones generate spotlights, whereas blocks and spheres generate positional lightsources. Light emitting facets generate directional lights. Rendering devices with global illumination models, e. g. Radiance, model more precisely the light distribution according to the shape of the primitive.

Quality Attributes
Quality attributes determine the visual quality of rendering primitives and scenes. Shading attribute objects define the shading method used to render primitives. Tessellation attribute objects define how fine analytic primitives are approximated (if the rendering devices cannot accept them in their parametric form). Jagged edges of lines and polygonal objects can be smoothed by antialiasing objects. Fog attribute objects add atmospheric effects to scenes. Not all renderers pay attention to quality attributes, e.g. ray tracers can ignore tessellation attributes because they draw analytic primitives in the best quality possible.

Cache Attribute
Cache attributes optimize the rendering process. They specify which of the rendering primitives are cached in the rendering device. For instance, OpenGL compiles cached rendering primitives in display lists. PEX rendering devices cache polyhedral approximations of all analytic primitives. The rendering protocol defines additional commands to control the object cache explicitly.

Transformation Attribute
The geometry of primitives is transformed by transformation attributes. Before primitives are rendered they are transformed by the current modeling and transformation matrix (CMTM). The CMTM consists of a local and a global 4x4 homogenous matrix. VRS defines several transformation classes such as scaling, translation, reflection, rotation, and general transformations matrices. CMTMs are used to build nested modeling coordinate systems. The world coordinate system is given by the matrix product of the global and the local transformation matrix.

Projections
Projection attributes transform the world coordinate system to the normalized projection coordinate system, i.e. they map the three-dimensional model space to two-dimensional screen pixels. VRS supports several types of projections:

- View: general projection specification [3] by view reference point, view plane normal, view up vector, projection reference point, view plane width and height, view plane positions, projection type (parallel or perspective).
- Camera view: Camera-model oriented view specification based on look-at point, eye point, distance, and field of view [14].
- Polar view: view specification in polar coordinates, i.e. azimuth and altitude.
- Orthogonal view: orthogonal projection.

Projection attributes maintain a view transformation matrix. VRS delegates the evaluation of projections to the rendering devices. Some rendering devices can map the parameters directly to rendering commands, others compute their own viewing matrices based on this information. Projections are elementary data types of the rendering layer. The modeling layer may provide high-level camera objects which are implemented based on projection attributes.

4 Design Considerations

4.1 Lightweight Objects

Primitives, transformations, and attributes are modeled redundancy-free, i. e. they contain only data which their type defines, and do not store context information such as transformation matrices or rendering devices. For example, spheres define their midpoint and radius, but do not define color or material properties. Instead, we supply context information through rendering devices or as arguments in operations. This approach leads to *lightweight objects*, i. e. objects which are minimal in their internal representation and which can therefore be used in large numbers [1]. They represent abstract graphical data types and can be used in the whole application comparable to integer or floating point data types since they do not involve any storage overhead. Increasing the fineness of the object-orientation increases the extensibility and reusability of the system. New attributes and new primitives can be derived from existing classes with minimal overhead.

To reduce transfer and conversion costs, rendering objects can be shared. Whenever an object references a shareable object, it increments the reference counter of this object. When the object dereferences the shareable object, it decrements its reference counter. If a shareable object is unreferenced, it is destructed automatically. Shareable objects provide a simple and efficient garbage collection. Objects of the modeling layer typically create and reference its rendering objects.

All rendering objects are persistent, i.e. they define operations to write themselves in (C++) streams textually, and to retrieve objects from streams. This persistency is used by objects of the modeling layer and facilitates their own persistency management.

4.2 Object Factories

Decomposing 3D graphics into fine-grained objects adds an additional amount of complexity through the high number of classes. To facilitate the access to rendering objects, VRS provides object factories. By *object factory* we mean classes which create objects of other classes. They summarize object constructors and provide a convenient interface to class hierarchies. The VRS object factory supplies methods to construct frequently used 3D objects. For example, it provides methods to create cylinders, cones with apex, and open tubes; all these objects are cone primitives but they can be created easier with the factory instead of calling the more complicated cone primitive constructor. Fig. 8 shows part of the VRS factory.

Object factories take care of persistency. They maintain a dictionary of known rendering classes and constructors to create default objects. Also rendering objects define persistency operations, i.e. they are able to write themselves into streams respectively they can retrieve their state from streams.

```
class VRSFactory : public Factory {
public:
  //shapes
  RCone* cylinder(Vector p, Vector q, float r);
  RCone* tube(Vector p, Vector q, float radius);
  RCone* cone(Vector base, Vector apex, float r);
  RBlock* cube(Vector center, float length);
  ...
  // apearance
  RColor* rgb(float, float, float);
  RColor* hsv(float, float, float);
  RFacetStyle* hollow();
  RShading* smooth();
  RShading* flat();
  ...
  // transformations
  RXform* scale(float s);
  RXform* scale(float sx, float sy, float sz);
  RXform* rotate(float angle, Vector axis);
  ...
  // rendering devices
  IORenderer* opengl();
  IORenderer* pex();
  IORenderer* xgl();
  Renderer* radiance();

  // persistency
  RenderObject* read(istream&);
  void write(RenderObject*, ostream&);

  void register_class(
   const char* classname, Constructor
  );
  //...
};
```

Figure 8: The VRS Object Factory.

Object factories register rendering classes, i.e. if a persistent rendering object is retrieved, the factory recognizes the object's class and constructs a default object. Then it delegates the retrieval process to that object. All VRS classes are registered in the VRS factory; to add application specific rendering classes, we register them at run time.

5 Example: The Landscape of Honolulu

In this example we build a VRS application which visualizes landscapes defined by triangle sets. First, we create the landscape application class 'XLandScape'. This class is derived from 'XVRS', a framework class which provides the user interface. The redraw method specifies the rendering commands applied to the virtual rendering device which is passed by XVRS as argument of the render method. An 'XLandScape' objects contains a landscape surface object. The render method delegates most of the visualization to this object.

Next, we define the landscape surface class. Landscape surface objects store the geometry of landscapes and their properties (e.g. water level and course of rivers). If a landscape surface object is constructed, it reads the geometry data file and maps this information to rendering primitives and rendering attributes. The mapping can be arbitrarily complex. ‘LandScapeSurface::render’ defines how the primitives and attributes are evaluated by virtual rendering devices. If an application object requests its landscape surface object to visualize itself, the landscape object sends its rendering objects to the virtual rendering device (passed as argument).

In the example, we use a triangle set to store the surface geometry. According to the position of the vertex and the slop of the triangle the vertex belongs to, we determine the vertex color of the vertex.

Finally, the main program consists in constructing an landscape object. The framework defines the ‘run’ method which starts the interface. The ‘XLandScape::redraw’ method is called implicitly if the window is exposed. In Fig. 9, part of the landscape of Honolulu is visualized.

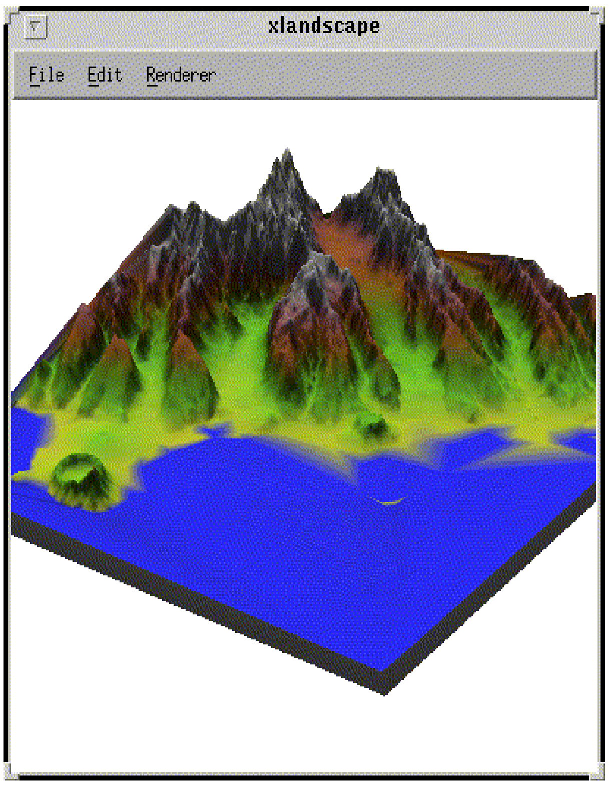
```
class XLandScape : public XVRS {
public:
 XLandScape(int argc, char** argv);
 virtual void redraw(Renderer&);
private:
 VRS<LandScapeSurface> surface_;
 // more primitives and attributes
};
   void XLandScape::redraw(Renderer& r) {
    r.begin_rendering();
    // push view, set background and lights
    surface_->render(r);
    r.end_rendering();
   }

class LandScapeSurface : public SharedObject {
 LandScapeSurface(
  char* file, Vector origin, Vector volume
 );

 virtual void render(Renderer&);

 // management of surface properties
 void set_water_level(float);
 void set_beach_level(float);
 //...
private:
 VRS<RTriangleSet> surface_;
 VRS<RReflection> reflection_;
 float water_level_;
 // more properties
};
   LandScapeSurface::LandScapeSurface(...) {
   // read, scale and translate the raw data
   // convert the data to triangle sets
   // calculate vertex normals and vertex colors
   }

   void LandScapeSurface::render(Renderer& r) {
    r.push(reflection_);
    r.render(surface_);
    r.pop(reflection_);
   }
```



```
int main(int argc, char** argv) {
 XLandScape L(argc, argv);
 L.run();
 return 0;
}
```

Figure 9: VRS Landscape Application. Part of the LandScape of Honolulu.

6 Implementation Details

VRS embeds OpenGL, PEX, XGL, and Radiance as rendering devices. Graphics package specific features are modeled by subclassed attribute classes. These rendering device classes support all core rendering primitives and attributes.

Mapping Primitives and Attributes

The abstract graphical data types must be converted into low-level rendering commands. PEX and XGL require their specific transfer format which involves a conversion overhead. No rendering device can map extruded primitives, rotational primitives, and the analytic primitives. These primitives are converted automatically by the rendering device base class into lower-level primitives, and are cached internally. The PEX rendering device (PEX 5.1) must additionally convert point sets, triangle fans, and quad strips. It cannot map antialiasing attributes because it does not define an accumulation buffer. Also, it cannot map non-solid textures. The XGL rendering device (XGL 3.x) must emulate point sets and quad strips. It supports all core attributes. The OpenGL rendering device provides the most efficient and elegant mapping of primitives and attributes since OpenGL commands are elementary and do not require a specific transfer format.

Code Statistics

Table 1 shows the size of the VRS interface and implementation, and the size of the library headers of

Table 1: VRS Interface and Implementation

Lines of Code[a]	Class Headers	Implementations	Library Headers
VRS	3100	6000	---
PEX Rendering Device	90	2200	4400 (PEXlib.h)
XGL Rendering Device	90	1700	3400 (xgl.h)
OpenGL Rendering Device	90	2000	1400 (gl.h, glu.h)

a. Lines of codes are rounded.

standard packages. The VRS object-oriented interface is smaller and handier as the PEX and XGL interface; it is larger than the OpenGL interface, but VRS raises the level of abstraction as compared to OpenGL. VRS class headers cover simultaneously several different graphics packages, it expresses the commonality between different rendering packages to a high degree. New rendering packages can be integrated by approximately 2000 lines of code. The implementations of new rendering devices benefit from the VRS core classes. Particularly, the Renderer base classes takes care of the attribute management and provides tessellators for analytic primitives, extruded primitives, and rotational primitives.

Performance

As compared to applications directly based on standard graphics packages, VRS applications provide the same performance. Rendering devices optimize the rendering process through the following techniques:

- Attributes are evaluated depending on the type of the primitive object. Typically, only a subset of the attribute objects are applied to a rendering object. The evaluation of an attribute object is delayed until a rendering object is rendered which actually depends on the attribute object’s values.
- Rendering devices evaluate an attribute object only if its current attribute values are different from the attribute object applied before. Therefore, virtual rendering devices do not use the expensive attribute stacks provided by most graphics packages.
- Rendering devices have direct access to the data of primitive objects. To maintain the encapsulation, primitive objects define methods which export read-only references to their internal data. For example, the OpenGL rendering devices traverse the data of all vertex-based primitives sending these data directly to OpenGL commands.

- Rendering devices save normalization and assertion tests because rendering primitives guarantee that their geometry fulfils explicitly specified conditions. These conditions can be guaranteed since primitive objects are encapsulated completely.

Compared to non-object oriented implementations, using the knowledge stored in objects can even improve the performance of VRS applications compared to applications directly based on low-level graphics packages.

7 Related Work

We use the following criterions to compare graphics systems:

- Access to key object-oriented techniques.
- Portability across different rendering techniques.
- Abstraction level of the interface.

GEO++

One of the first object-oriented low-level graphics systems is GEO++ [8], a Smalltalk-80-based system which encapsulates parts of the functionality of PHIGS and GKS. The class library focuses on supporting object hierarchies. All graphical entities are modeled by objects. Thus, key object-oriented techniques such as inheritance and polymorphism, are available. Due to its experimental character, the class hierarchy does not cover 3D primitives and attributes; therefore the functionality is limited to 2D graphics and cannot be compared to VRS. GEO++ is portable within Smalltalk-80, but is not available for other graphics packages.

HOOPS

HOOPS [15], the hierarchical object-oriented programming system, is a portable subroutine library similar to PHIGS. Object-orientation refers to the support of hierarchically composed objects, but it is not implemented in an objected-oriented language and does not provide class hierarchies. Thus, HOOPS applications cannot take advantage of class interfaces, objects, polymorphism, and inheritance. Portability across all current hardware and software platforms is provided. HOOPS serves as high-level application programming interface to computer graphics. It cannot be integrated smoothly in object-oriented systems due to its declarative architecture and to the missing fine-grained access and control over graphical entities.

GRAMS

GRAMS, an object-oriented 3D graphics system, focuses on the object-oriented paradigm. It raises the level of abstraction for the application designer due to the logical division of graphical applications in three layers: application, graphics, and rendering layer. GRAMS defines formally the graphics layer which interacts with the application layer and the rendering layer. The rendering layer can be exchanged, but it is not based on general, abstract graphical data types as in VRS. In contrast to GRAMS, VRS defines an application and a modeling layer; classes similar to GRAMS graphical objects are part of the VRS rendering layer because these objects are atomic in the developer's perspective, but standardized for all rendering devices.

The criterion used in GRAMS to build the class hierarchy is the commonality in the internal representation of graphical objects. This leads to semantic inconsistencies and to not fully reusable class hierarchies since operations of base classes may contradict the semantic of the derived classes. For example, GRAMS models cones and cylinder in two sibling classes, VRS provides a general cone class, and represents cylinders as cone objects with equal top and bottom radius.

Graphical objects in GRAMS contain shape, transformation, and attribute information. Attributes are not modeled through an attribute class hierarchy. New attribute types cannot be derived or integrated in graphical objects. Storing transformation matrices and attributes in graphical objects increases the object size, and prevents objects from being shared and used in large numbers.

GROOP

GROOP, another object-oriented 3D toolkit, derives its basic metaphors from movies: stage, actors, light, and cameras. It separates rendering from modeling in two system components. However, it is implemented only for OpenGL. Due to the class hierarchy which is based on implementation considerations, GROOP cannot be ported easily to other rendering techniques. GROOP derives cylinders from triangle meshes assuming that renderers are not capable of drawing cylinders as analytic primitives (which is the case for all ray tracing and radiosity packages).

As in GRAMS there is one base class for all geometric objects which stores material properties and transformations. This implies the same consequences as discussed above for GRAMS. Nevertheless, the class hierarchy is fully accessible and can be used for deriving new types of geometric objects.

The abstraction level of GROOP is high, but the level of object-orientation is still coarse-grained. In VRS terms, GROOP defines a modeling layer for computer animation, but does not define a separate and object oriented rendering layer.

TBAG

TBAG [11] is based on a set of graphical abstract data types. It provides a high-level interface to interactive 3D graphics. Scenes are composed by values of graphical abstract data types instead of non-temporary objects. This is computationally expensive. To draw a scene, all graphical objects used must be generated due to TBAG’s functional approach. Compared to VRS, TBAG’s concept cannot be easily integrated into an object-oriented application model because the functional approach to describe models is not suitable for real world scenes which in general can be described easier in a declarative and hierarchical rather than a functional manner. TBAG does not support different rendering techniques.

8 Conclusion

We have presented an object-oriented virtual rendering system which embeds standard low-level 3D graphics packages in an extensible class library. To serve for all kinds of graphical applications, VRS standardizes the capabilities of graphics packages through abstract graphical data types which are used to build modeling or application objects, and virtual rendering devices which operate on these types. The VRS class hierarchy provides logically structured rendering objects including a rich set of geometric primitives, rendering attributes, transformations, and rendering devices. These objects are shareable and provide automatic storage management. The rendering language consists of only three commands: push and pop of attributes, rendering of primitives. New rendering packages can be easily integrated by deriving specialized rendering device classes. Individual rendering features are integrated by deriving new attribute and primitive classes. All rendering devices implement the rendering protocol in order to guarantee a minimal set of rendering operations. Additionally, rendering capabilities and costs can be inquired. VRS ensures extensibility and reusability by its open system architecture. VRS is implemented as a C++ toolkit integrating OpenGL, PEX, XGL, and Radiance. Currently, we extend the system by other packages including RenderMan [7] and POVRay[17].

References

[1] Paul Calder and Mark Linton. Glyphs: Flyweight objects for user interfaces. Proceedings of the *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Snowbird, Utah, pp. 92 - 10, October 1990.

[2] Parris K. Egbert and William J. Kubitz. Application Graphics Modeling Support Through Object Orientation.In *COMPUTER*, pp. 84 - 91, October 1992.

[3] James D. Foley et al. *Computer Graphics: Principles and Practice*, 2nd. Edition. Addison-Wesley, Reading, MA, 1990.

[4] Larry Koved and Wayne L. Wooten. GROOP: An object-oriented toolkit for animated 3D graphics. In *ACM SIGPLAN NOTICES OOPSLA ‘93*, 28(10):309-325, October 1993.

[5] Mark A. Linton, Paul R. Calder, and John M. Vlissides. The Design and Implementation of InterViews. *Proceedings of the USENIX C++ Workshop*, Santa Fe, New Mexico, November 1987.

[6] PHIGS+ Committee, Andries van Dam, chair. PHIGS+ Functional Description, Revision 3.0. *Computer Graphics*, 22(3), pp. 125 - 218 July 1988.

[7] Steven Upstill. The RenderMan Companion. A Programmer's Guide to Realistic Computer Graphics. Addison-Wesley, Reading, MA, 1990.

[8] Peter Wisskirchen. *Object-Oriented Graphics: From GKS and PHIGS to Object-Oriented Systems*. Springer-Verlag, Berlin, 1990

[9] Randi J. Rost, Jeffrey D. Friedberg, and Peter L, Nishimoto. PEX: A Network-Transparent 3D Graphics System. In *IEEE Computer Graphics & Applications*, pp. 14 - 26, July 1989.

[10] Solaris XGL 3.0.1 Reference Manual, SunSoft, Sun Microsystems, Inc., Mountain View, CA 94043, 1993.

[11] Conat Elliot, Greg Schechter, Richy Yeung, and Salim Abi-Ezzi SunSoft, Inc. TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications. In *Proceedings of SIGGRAPH '94*, pages 421-434.

[12] Gregory J. Ward. The RADIANCE Lighting Simulation and Rendering System. In *Proceedings of SIGGRAPH '94*, pp. 459-472.

[13] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading, MA, 1993.

[14] Josie Wenecke. *The Inventor Mentor. Programming Object-Oriented 3D Graphics with OpenInventor*, Release 2. Addison-Wesley, 1994.

[15] Joel Orr. HOOPS 3.0: Beyond 3-D. In *Byte*, Februrary 1994.

[16] James Rumbaugh. et al. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., 1991.

[17] POV-Ray Team, Persistence of Vision Ray Tracer (POV-Ray), Version 2.0, User's Documentation, 1993